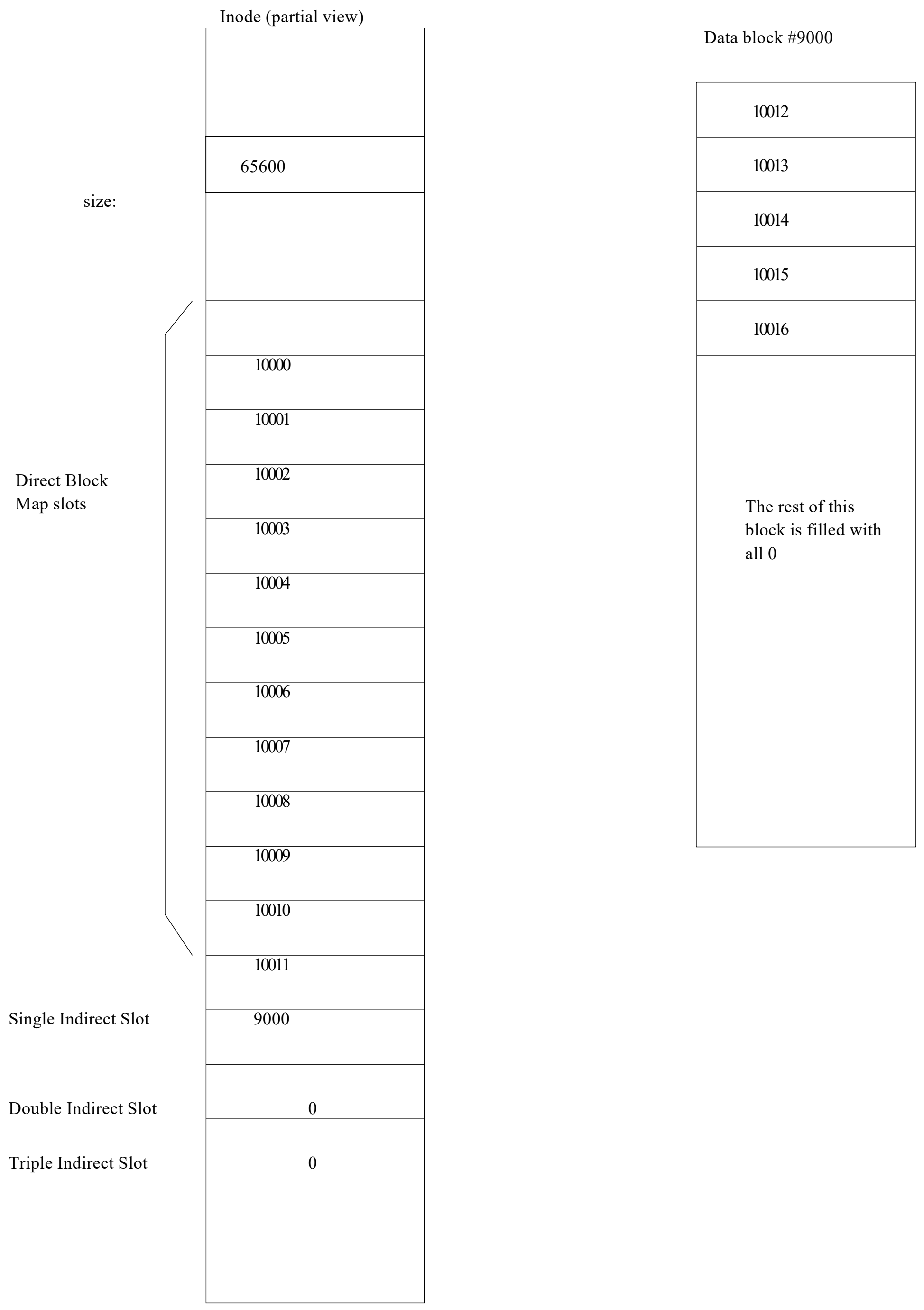
Allister Liu

Problem 1 -- File allocation

The following diagram depicts part of the information contained inside an inode which represents an ordinary file. Specifically, it shows the block allocation area of the inode and the size. Also shown is one data block which is being used as an indirect block. This is a Linux EXT3 filesystem and block sizes are the default 4K. *Note: all values in the diagram below are expressed in decimal (base 10)*



a)What is the number of the data block (express in decimal) which contains byte 4100 decimal (0x1004 hexadecimal)?
10001
b)Ditto, for byte 65535 dec (0xFFFF hex)
10015 – in data block #9000.
c)If instead of EXT3, this were an EXT4 filesystem using extent-based allocation, but the same blocks were allocated to the file in the same order, how many extent descriptors would be required? Where would the extent descriptor(s) be stored? Justify your answer.
One extent descriptor is needed. Because the block count is less than 2^15, and they are not fragmented.
d)Now, back to EXT3, and we modify the file as follows: we `lseek` to file offset 409600 (decimal) bytes and then `write` a single byte. How many additional data blocks does the kernel have to allocate? Justify your answer.
One additional data block is needed to be allocated. Despite offsetting by 100 4K data blocks, there isn't data in between, so we only need one extra data block to write the single byte of data.

Problem 2 -- Exploratory Questions

Answer each of the following with a paragraph or two **in your own words**. After you read the lecture notes, you'll have enough material to answer. **Don't just copy the lecture notes verbatim**, use your own words.

A) In browsing the inode table of an EXT3 filesystem I observe an inode with inode type `S_IFDIR` and `nlink`==4. What does this link count tell us about the contents of this directory inode?

The inode type S_IFDIR indicates that the inode represents a directory. nlink == 4 means that there are 4 pathnames linking to the inode, including one pointing to the current directory (.) and one pointing to its parent directory (..), and two more linking to its children. If the directory indicated by this inode happens to be the root directory, the .. link will also point to itself.

B) There is a directory say `/home/pcooper/stuff` which has about 50,000 files. The first time I `open("/home/pcooper/stuff/12345.txt",O_RDONLY)` it takes a "long time" but when I open the same file again a few seconds later the `open` system call returns much faster. Explain.

This is because of concept of caching - the assumption that if you do something once, you are likely to do it again, or something very much like it or near it, in the near future. Thus, when you first opened the file 12345.txt, a live copy of its inode is kept in the kernel memory. The given directory /home/pcooper/stuff is physically stored in a mass storage device (e. g. hard drive), that means the kernel will need to access the hard drive to retrieve data (first open() system call), which is considerably much slower than accessing the kernel memory (when you make open() system call again).

C) We have an EXT3 filesystem with journalling turned on, using mode `data=ordered`. Some schlemeil trips over the power cord while the system has filesystem activity. When the system powers back up, `fsck` examines the journal and finds this:

```
--BEGIN TRANSACTION ID#1234--
Copy of block containing inode #9999 type==FILE size==0 nlink==1
Copy of block containing entry 9999 in free inode bitmap, showing i9999 allocated
Copy of block containing new directory entry referring to ino9999
Copy of block containing ino100 (parent of i9999) with updated mtime
--COMMIT TRANSACTION ID #1234--
--BEGIN TRANSACTION ID#1235--
Copy of block containing free block bitmap entry for block #5555, showing d5555
allocated
Copy of disk block containing inode #9999, now size==4096, mtime updated
--COMMIT TRANSACTION ID #1235--
```

i) From examining the journal, what does it appear was happening?

inode #9999 is marked non-free (changing its type to file, set the size to 0, and nlink to 1).
inode #9999 is marked non-free in the inode bitmap.
Path component name and the inode number 9999 are written to a directory.
The parent of i9999 is updated (nlink +1).
Block #5555 is marked as in-use in free block bit map.
Space of a size of 4K is allocated in block maps for inode #9999.

ii) Before we "replay" the above two journal entries, can we say what state inode #9999 is on the disk?

Yes, because the file is set to data=ordered mode, which metadata are written into journal after data writeback, so by following the journal (like we did in i) we can find out the state of #9999 (which is described in previous question).

iii) After the "replay" would there be any perceptable corruption of the filesystem?

No, because the filesystem is in ordered mode, where the metadata are updated to the journal after the data are written back. From looking at the journal, the inode #9999 is correctly linked and properly configured, and all the data are written back before the system lost power. So, I don't see any possible corruption of the filesystem.

iv) Reconsider question (iii) but this time assume the filesystem journal mode was set to `data=writeback`

If the system is set to writeback journal mode, a possible corruption of the filesystem. Because in writeback mode, the write-back of data blocks is not synchronized at all with metadata writes to the journal. So when the system regains power and looks at the journal, it appears that the file is properly created and 4K data block is allocated for it; however, since the writeback is not at all synchronized with the metadata, we do not know which action described in the journal is actually completed, thus we may see garbage.

D) I mount a filesystem:

```
# mount /dev/sdb1 /mnt/usbdrives/volume01
# ls -ladi /mnt/usbdrives/volume01
```

Where `/mnt/usbdrives/volume01` was an empty directory within the root filesystem (aka root volume) and is inode #10. What is the inode # reported by the ls command? Explain your answer.

The inode # reported by the ls command should be #2. Executing the command mount /dev/sdb1

/mnt/usbdrives/volume01 makes /mnt/usbdrives/volume01 (to become) the mount point. Despite its inode #10 in the root filesystem, it is replaced with the root inode #2 of the mounted /dev/sdb1, obscuring the original inode #10.

E) A user runs the command `mv /A/B/F1 /A/C/F2;` where `F1` is a fairly large file. This command runs very quickly. Then the user runs `mv /A/C/F2 /A/Z/F3;` but this takes quite a while to run. Why might that be the case?

One possible explanation can be that the directory /A/C and /A/Z are not in the same volume (device), while /A/B and /A/B are in the same volume. When the mv(1) command is used to move a file within the same volume, it uses the rename(2) function, which creates a hard link (link(A/B/F1 A/C/F2)) then removes the old link to F1. This leaves the file's content untouched, but rather changed the pathname to access it. Meanwhile, moving a file to a different volume using mv, rename(2) cannot be used because a hard link cannot be made from one volume to another; therefore it does the equivalent of cp (A/C/F2 A/Z/F3) then rm (A/C/F2) to actually copy every byte of the file to the new path then deleting the old. Comparing to simply using rename(2), the later method would take significantly longer.

SUBMISSION: Submit your answers to problems 1 / 2 in the format of either a PDF or plain text document.